Parallel Computing - Work Assignment 1

Optimising a molecular dynamics' simulation code

October 23, 2023

José Fonte
Universidade do Minho
a91775

Bernard Georges
Universidade do Minho
pg53698

Abstract

This assignment focus on optimising a simple molecular dynamics' simulation applied to atoms of a given gas. The work group used several optimization techniques to decrease the running time from *250 secs* to *5.35 secs*. The following report explains the steps and rationale of the group that led to 97.86% improvement over the original running time.

1 | Introduction

The following report describes the work developed by the team in analizing and optimising an inneficient program provided by the Professors. The given code is a simple molecular dynamics' simulation applied to atoms of a given gas - it uses Newton's Law and Lennard Jones potential to describe the interactions among two particles (force/potential energy).

With the goal being lowering the running time, we have the following equation:

$$\mathrm{CPU}_{\mathrm{time}} = \#\mathrm{Inst} * \mathrm{CPI} * \mathrm{Clk}_{\mathrm{cycles}}$$

Given that the $\mathrm{Clk}_{\mathrm{cycles}}$ is 2.50GHz, in this case the CPU is an Intel(R) Xeon(R) CPU E5420 @2.50GHz, the goal is to decrease the #Inst as much as possible without comprimissing CPI - dependent on the machine code produced and CPU performance.

2 | Optimization journal

The methodology followed by the group was of continuous analisys and optimization, the following chapters describe the process the group went through to achive the best running time of *5.35 secs*.

2.1 | Big O | Code Analysis

The Big O analysis enables us to identify the space and time complexity of a piece of code and do further optimizations if possible. Throughout our analysis, the group noticed that various loops complexity could be reduced using well-know Loop-Level Parallelism techniques such as Loop Unrolling and some flags mentioned in 2.4 | Flags Used.

Potential Improvements :

- *potential()* : $O(3n^2) \to O(n^2)$
- *computeAccelerations()* : $O(6n^2) \to O(n^2)$
- *velocityVerlet()* : $O(7N) \to O(3N)$
- *initializeVelocities()* $O(15N) \to O(3N)$

2.1.1 | Loop Unrolling

Loop unrolling executes multiple loop iterations in a single pass, leading to an increase in performance due to:

- reduced loop overhead - lowering conditionals jumps, control variables increment and mispredictions in jumps.
- increased LLP - allowing the processor to execute instructions concurrently every pass taking full advantage of multiple execution pipelines.

The implementation of this technique throughout the code can be seen in attachments : Listing 1 and Listing 2, which shows the code before and after loop unrolling in the *potential()* function.

Even though none of the improvements reduce the complexity in an order of magnitude (*E.g. :* $O(n^3) \to O(n^2)$), the way the code works makes it that a small improvement has a compounding effect, leading to major improvements, better described in "Results Comparison".

2.2 | Gprof Results | Profilling

The *gprof* tool provides a detailed breakdown of where the majority of CPU cycles are being consumed, allowing us to pinpoint were our atention to optimization needs to be.

As shown in Figure 5 our main focus at thatmoment had to be the *Potential()* function.

2.2.1 | Simplifying Math Functions

After analising the function code the group noticed that the math used could be simplified.

$$\begin{cases} \mathrm{rnorm} = \sqrt{\mathrm{r2}} \\ \mathrm{quot} = \frac{\mathrm{sigma}}{\mathrm{rnorm}} \end{cases} \Rightarrow \begin{cases} \mathrm{quot} = \frac{\mathrm{sigma}}{\sqrt{\mathrm{r2}}} \end{cases}$$

$$- - - - -$$

$$\begin{cases} \mathrm{term2} = \mathrm{quot}^6 \\ \mathrm{term1} = \mathrm{quot}^{12} \end{cases} \Rightarrow \begin{cases} \mathrm{term2} = \frac{\mathrm{sigma}^6}{\sqrt{\mathrm{r2}}^6} \\ \mathrm{term1} = \frac{\mathrm{sigma}^{12}}{\sqrt{\mathrm{r2}}^{12}} \end{cases} \Rightarrow \begin{cases} \mathrm{term2} = \frac{\mathrm{sigma}^6}{\sqrt{\mathrm{r2}}^3} \\ \mathrm{term1} = \frac{\mathrm{sigma}^{12}}{\mathrm{r2}^6} \end{cases}$$

$$\Rightarrow \begin{cases} \mathrm{term2} = \frac{\mathrm{sigma}^6}{\sqrt{\mathrm{r2}}^3} \\ \mathrm{term1} = \frac{\mathrm{sigma}^6}{\sqrt{\mathrm{r2}}^3} * \frac{\mathrm{sigma}^6}{\sqrt{\mathrm{r2}}^3} \end{cases} \Rightarrow \begin{cases} \mathrm{term2} = \frac{\mathrm{sigma}^6}{\sqrt{\mathrm{r2}}^3} \\ \mathrm{term1} = \mathrm{term2} * \mathrm{term2} \end{cases}$$

Using the equations presented we can simplify the code in Listing 3 to the one in Listing 4. This procedure helped reduce the amount of multiplications and removed the `sqrt()`and `pow()` functions which take a heavy toll on the ALU performance. As mentioned previously, because of the compounding effect of this optimization and all others, these small changes had a big consequence on the #Inst of our program. Besides that calling external libraries can cause various overhead because of data transfer serialization and the function call overhead.

- *transfer serialization*: when your program needs to pass large data structures or complex objects to external library functions, the serialization and deserialization processes can be computationally expensive.

With this changes inplace the group, once again, used the *gprof* tool that now indicated *computeAccelerations* as the function to tackle next.

2.3 | Matrixes Analysis | Code Analysis

After analysing the matrixes the group noticed that some functions run similar workloads, creating a large amount of repeated instructions over the program's execution time. Therefore the functions *computeAccelarations()* and *potential()* were joined into *joinedPotentialComputeAcc()* and the functions Kinetic() and *MeanSquaredVelocity()* were joined into *joinedKineticMSV()*.

2.3.1 | Functions Fusion

The first coupling *joinedPotentialComputeAcc()* was due to both acessing the matrix `r[i][j]`. Some small changes were made to the code to accomodate this fusion, the one that should be noticed is *Potential()* loop was changed to avoid doing double the calculations, something the group noticed on analysis - this means the nested for loop now goes from $i \to N$ and the potential doubles so it is $\text{PE} += 8 * \text{epsilon} * (\text{term1} - \text{term2})$.

The second pairing *joinedKineticMSV()* was due to both acessing the matrix `v[i][j]` and doing some calculations after. Another reason was because there was data independency which means their variables don't need the others calculations which enables parallel execution.

The couplings helped reduce the amount of times the code needed the matrixes and the amount of loop needed to do so by calculating shared values that in the end were used in equations to generate the goal. This helped in reduce the loop overhead, amount of memory access and computation of the equations by atleast N times for each fusion being the fusion of the *Potencial* and *computeAccelerations* the most noticable of all the changes.

2.4 | Flags Used

The last step of optimization was implementing a few flags so the compiler produces better machine code, each one of these plays a role in optimizing and enhancing the performance of code, making it more efficient for specific hardware.

- `-Wall` : used to enable a comprehensive set of compiler warning messages, such as uninitialized variables, unused variables, logical errors and other problems.
- `-pg` : required by *gprof* which is a profilling tool that allows you to gather performance data about your program.
- `-O3` : flag designed to maximize the execution speed of your code by applying aggressive optimization techniques, it checks all optimization flags available, including `-ftree-vectorize` that enables the compiler to vectorize loops and utilize SIMD instructions.
- `-msse4` : specifies that the compiler should generate code using SSE4 instructions which allows operations with 128bits vectors.
- `-mavx` : instructs the compiler to generate code that uses AVX instructions, which offer even more advanced SIMD capabilities.
- `-mfpmath=sse` : specifies that the compiler should use SSE math instructions for floating-point operations.
- `-march=x86-64` : sets the target architecture to x86-64 (64-bit x86 architecture), optimizing code generation for 64-bit systems.

2.5 | Common Compiler Optimizations

Some Common Compiler Optimizations like Loop Unrolling, Fusion & Reduction in Strength were already mention in there previous contexts but other optimizations were also implemented such as Dead Store Elimination & Constant folding.

2.5.1 | Dead Store Elimination

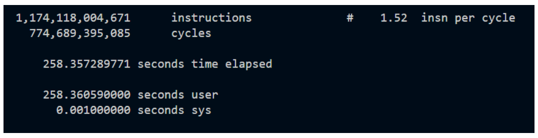
Due to the -Wall compiler flag and some code analysis, multiple variables were flaged as being unused, being some examples of the ones removed:

- `trash[10000]`
- `F[MAXPART][3]`
- `*infp`
- all the loose variables from loop unrolling
- reducing number of chars in `prefix[1000], tfn[1000], ofn[1000], afn[1000]` to 32 chars.

3 | Results Comparison

At this final stage the group decided on making some final observations about the improvents in each phase of development using the *srun* tool to show the time, instructions and clock cycles for every iteration.

3.1.1 | Loop Unrolling



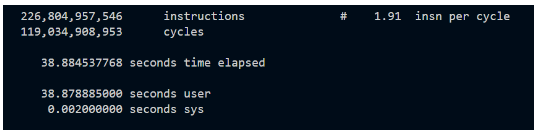
```
1,174,118,004,671      instructions              #    1.52  insn per cycle
  774,689,395,085      cycles

    258.357289771 seconds time elapsed

    258.360590000 seconds user
      0.001000000 seconds sys
```

Figure 1: Result after loop unrolling

3.1.2 | Symplifying Math Functions



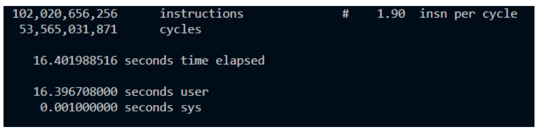
```
 226,804,957,546      instructions              #    1.91  insn per cycle
 119,034,908,953      cycles

    38.884537768 seconds time elapsed

    38.878885000 seconds user
     0.002000000 seconds sys
```

Figure 2: Result after symplifying math functions

3.1.3 | Functions Fusion



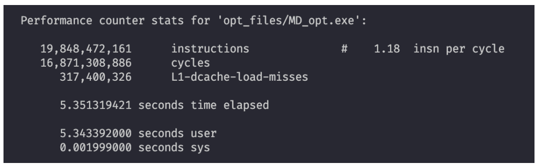
```
102,020,656,256      instructions              #    1.90  insn per cycle
 53,565,031,871      cycles

   16.401988516 seconds time elapsed

   16.396708000 seconds user
    0.001000000 seconds sys
```

Figure 3: Result after functions fusion

3.1.4 | Final Result with Flags Used



```
 Performance counter stats for 'opt_files/MD_opt.exe':

    19,848,472,161      instructions              #    1.18  insn per cycle
    16,871,308,886      cycles
       317,400,326      L1-dcache-load-misses

       5.351319421 seconds time elapsed

       5.343392000 seconds user
       0.001999000 seconds sys
```

Figure 4: Final Result with all optimizations

4 | Conclusion

On our final note we believe the results were good and we are very happy with the amount we learned and improved a long the project. We particularly enjoyed learning about the compiler optimization techniques as well as avoiding data dependencies. We believe that we could have explored more of the vectorization of the code and further improvents through the analysis of the assembly code.

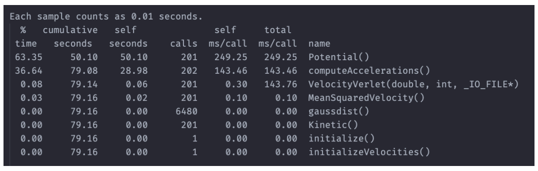
5 | Attachments:

```
for (k=0; k<3; k++) {
    r2 += (r[i][k]-r[j][k])*(r[i][k]-r[j][k]);
}
```

Listing 1: Code from the *Potential()* before optimizing

```
r2 = (r[i][0]-r[j][0])*(r[i][0]-r[j][0])
+ (r[i][1]-r[j][1])*(r[i][1]-r[j][1])
+ (r[i][2]-r[j][2])*(r[i][2]-r[j][2]);
```

Listing 2: Code from the *Potential()* after optimizing



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 63.35     50.10    50.10      201   249.25   249.25  Potential()
 36.64     79.08    28.98      202   143.46   143.46  computeAccelerations()
  0.08     79.14     0.06      201     0.30   143.76  VelocityVerlet(double, int, _IO_FILE*)
  0.03     79.16     0.02      201     0.10     0.10  MeanSquaredVelocity()
  0.00     79.16     0.00     6480     0.00     0.00  gaussdist()
  0.00     79.16     0.00      201     0.00     0.00  Kinetic()
  0.00     79.16     0.00        1     0.00     0.00  initialize()
  0.00     79.16     0.00        1     0.00     0.00  initializeVelocities()
```

Figure 5: gprof profilling on the base code.

```
rnorm=sqrt(r2);
quot=sigma/rnorm;
term1 = pow(quot,12.);
term2 = pow(quot,6.);
```

Listing 3: Code from *Potential()*

```
sigma6 = sigma*sigma*sigma*
        sigma*sigma*sigma
quot = sigma6 / (r2);
term2 = quot * quot * quot ;
term1 = term2 * term2;
```

Listing 4: Code simplified from *Potential()*